A Decision Procedure for Regular Expression Equivalence in Type Theory*

Thierry Coquand and Vincent Siles

University of Gothenburg
{coquand,siles}@chalmers.se

Abstract. We describe and formally verify a procedure to decide regular expressions equivalence: two regular expressions are equivalent if and only if they recognize the same language. Our approach to this problem is inspired by Brzozowski's algorithm using derivatives of regular expressions, with a new definition of finite sets. In this paper, we detail a complete formalization of Brzozowki's derivatives, a new definition of finite sets along with its basic meta-theory, and a decidable equivalence procedure correctly proved using Coq and Ssreflect.

Introduction

The use of regular expressions is common in programming languages to extract data from strings, like the `scanf` function of the C-language for example. As shown in recent works [4,11] the equational theory of regular expressions can also be important for interactive theorem provers as providing a convenient tool for reasoning about binary relations. The fundamental result which is used there is the *decidability* of the problem whether two regular expressions are *equivalent*, i.e. recognize the same language, or not.

The purpose of this paper is to represent in type theory the elegant algorithm of Brzozowski [5] to test this equivalence. In an intuitionistic framework such as type theory, this in particular amounts to show that the equivalence between two regular expressions is a decidable relation. For this, we define in type theory a boolean valued function corresponding to Brzozowski's algorithm, and we show that this function *reflects* [18] equivalence: this function returns true on two regular expressions if and only if they are equivalent.

Brzozowski's algorithm has already been formally investigated but it never has been *completely* proved correct: in [11], the authors did not proved formally the termination of their algorithm, and in [1], the authors did not finished the proof of correctness of the procedure. In this paper, we describe a *complete* formalization of Brzozowski decision procedure based on derivatives of regular expressions.

In order to achieve this formal representation, we introduce a new definition of finiteness in type theory, which may have an interest in itself. This definition

* The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

J.-P. Jouannaud and Z. Shao (Eds.): CPP 2011, LNCS 7086, pp. 119–134, 2011.

is not equivalent constructively to the usual definition, which expresses that one can list all elements in this set. (Intuitively this new definition expresses that if we keep removing elements from a finite set, this will stop eventually.) We believe that this notion is useful to express in type theory algorithms relying on the computation of least fixed point of finite sets (see Coro. 1), such as the computation of minimal automata or the computation of a deterministic automaton associated to a non deterministic automaton.

In Sect. 1, we describe a new definition of finite sets called *inductively* finite sets, along with some basic properties required by the decision procedure correctness. In Sect. 2, we recall the definition of regular expressions and the notion of Brzozowski derivatives. The third section is dedicated to Brzozowski's proof that the set of derivatives is finite (*inductively* finite in our case). The decision algorithm for equivalence is then defined by recursion over this proof of finiteness and we discuss its representation in type theory. This uses an elegant idea of Barras for representing in type theory functions defined by well-founded recursion, which keeps logical soundness while having a satisfactory computational behavior. The last section presents some test cases. The whole development has been formalized[1] using Coq [7] and Ssreflect [18], and can be found at [8].

1 Finite Sets in Type Theory

1.1 Informal Motivations

The notion of *finiteness* is basic in mathematics and was one of the first notion to be formalized by Dedekind and Frege. A 1924 paper by Tarski [19] describes different possible axiomatizations in the framework of Zermelo set theory. One definition comes from Russell and Whitehead [17]: a subset of a set is finite if it can be inductively generated from the empty set by the operation of adding one singleton. A set A is then finite if A itself is a finite subset. Tarski shows that this is equivalent to another definition, in the framework of classical set theory: a set is finite if the relation of strict inclusion on subsets is well-founded. (By taking complement, this is equivalent to the fact that the relation $X \supsetneq Y$ is well-founded, i.e. there is no infinite sequences $X_0 \subsetneq X_1 \subsetneq X_2 \subsetneq \dots$) On the other hand, these two definitions are only equivalent to Dedekind definition (a finite set is such that any endomap which is injective is a bijection) in presence of the axiom of choice.

In intuitionistic frameworks, the most commonly used definition seems to be the one of being Kuratowski finite [10], which is a small variation of Russell-Whitehead definition: a subset is finite if it can be inductively generated from the empty set, the singleton and the union operations. In type theory, this definition takes an equivalent more concrete form: a subset is finite if and only if it

[1] There are no axioms neither in this development nor in Ssreflect libraries. However, due to a flaw in the Print Assumptions command, one might think there are. This command considers definitions made opaque by signature ascription to be axioms, which is not the case.

can be enumerated by a list. The situation is complex if we consider sets with a non necessarily decidable equality. In this paper however, we limit essentially ourselves to *discrete* sets where the equality is decidable. With this extra assumption, to be Kuratowski finite is equivalent to be in bijection with a set N_k where N_k is defined recursively by $N_{k+1} = N_k + N_1$ where N_0 is the empty type and N_1 the unit type. So, given the information that A is Kuratowski finite, we can compute from it the cardinality of A, and, in particular, we can decide if A is empty of not.

In this paper, we explore another inductive definition of finiteness: a set A is *noetherian* (or *inductively finite*) if and only if for all a in A the set $A - \{a\}$ is noetherian. Intuitively, if we keep picking distinct elements in A, eventually we reach the empty set. It should be intuitive then that if A is Kuratowski finite then A is noetherian (by induction on the cardinality of A), but also that, only from the information that A is noetherian, one cannot decide if A is empty or not. So to be noetherian is intuitionistically weaker than being Kuratowski finite. (See the reference [9] which analyzes these notions in the framework of Bishop mathematics.) We have shown formally that to be noetherian is equivalent in type theory to a variation of Tarski definition: A is noetherian if and only if the relation $X \supsetneq Y$ is well-founded on subsets of A given by lists. Independently of our work, corresponding definitions of finiteness have been recently considered in the work [2].

1.2 Inductively Finite Sets in Type Theory

From now on, we use predicates over a given type A to represent subsets. Predicates over A are `Prop`-valued functions, the universe of propositions of Coq. Given a binary relation R and an element x of type A, $R\ x$ represents the set of the elements $\{y : A \mid R\ x\ y \text{ holds }\}$.

Given a type A, a set E and a binary relation R over A, we say that **Bar** $R\ E$ holds if and only if for any x in E, **Bar** $R\ (E \cap (R\ x))$ holds. This is an inductive definition[2] , which expresses intuitively that we can not find an infinite sequence $x_0,\ x_1,\ x_2, \ldots$ of elements satisfying E and such that we have $R\ x_i\ x_j$ if $i < j$.

This definition is closely connected to the notion of well-quasi-ordering [16]. Indeed R is a well-quasi-ordering on E if and only if it is transitive and decidable and its complement R' is such that **Bar** $R'\ E$ holds. If we start from a type A with a decidable equality $=_A$ then we define E to be (inductively) finite if **Bar** $(\lambda\ x.\lambda\ y.\neg\ (x =_A y))\ E$. Intuitively, it expresses that for any infinite sequence $x_0,\ x_1,\ x_2, \ldots$ there exists $i < j$ such that $x_i =_A x_j$.

As explained above, the usual definition of finite (or "Kuratowski finite") is that we can list all elements in E (see Fig. 2). It can be shown that E is inductively finite if it is finite, but the converse does not hold. Therefore, we capture more sets with this definition, but in general, it is not possible to describe an inductively finite set as the list of its elements.

[2] This is a particular case of *Bar* induction [16], and we kept the name.

```
Variable A:Type.
Definition gset := A → Prop.
Definition grel := A → A → Prop.

Inductive Bar (R: grel A ) (E: gset A ) : Prop :=
  | cBar : (∀ x:A, E x → Bar R (Intersection E (R x))) → Bar R E.

Definition IFinite (R : grel A) (E : gset A) := Bar (neq R) E.
```

Fig. 1. Definition of **Bar**

Lemma 1. Basic properties of inductively finite sets

- *If* **Bar** R F *and* $E \subseteq F$ *then* **Bar** R E.
- *If* **Bar** R E *and* **Bar** R F *then* **Bar** R $(E \cup F)$.
- *If* **Bar** R E *and* **Bar** S E *then* **Bar** $(R \cup S)$ E.
- *If* **Bar** R E *and* **Bar** S F *then* **Bar** $(R \times S)$ $(E \times F)$, *where* $(E \times F)$ (x,y) *means* $E\ x \wedge F\ y$, *and* $(R \times S)$ (x_0,y_0) (x_1,y_1) *means* $R\ x_0\ x_1 \vee S\ y_0\ y_1$.

The proof of the last point is essentially the same as the proof that a product of two well-quasi-ordering is a well-quasi-ordering [16].

For any set E over a type A, a list of elements of E can be interpreted as a finite subset of E. Any list defines a subset of E by defining a membership function InA:

- InA $[\,]\ x$ is always false
- InA $(hd\ ::\ tl)\ x$ holds if and only if $x =_A hd$ or InA $tl\ x$.

Using this membership function, we can describe in type theory what it means to be Kuratowski finite. E is Kuratowski finite when there is a list X that enumarates the elements of E:

$$\exists X : [A], \forall x : A, E\ x \leftrightarrow \mathsf{InA}\ X\ x$$

If A is a type with decidable equality $=_A$, InA is a decidable predicate, and we can define a decidable equality $=_{[A]}$ on the type $[A]$ of lists (also written eql in the code) such that $X_0 =_{[A]} X_1$ holds if and only if X_0 and X_1 represent the same subset of E. If X is of type $[A]$, we define $[E]\ X$ (or gpred_list $E\ X$) to mean that all elements in X satisfy the predicate E.

Since we are working with an abstract equality over the type A, a natural condition on E is to require it to be *compatible* with the equality over A, that is for all x, y such that $x =_A y$ and $E\ x$ holds, then $E\ y$ also holds.

Proposition 1. *(Bar_gpred_list) If E is inductively finite on A and compatible then [E] is inductively finite on [A].*

This can be expressed as the result that the collection of all subsets (given by lists) of an inductively finite set is inductively finite. The proof is reminiscent of the constructive proof of Higman's Lemma about well-quasi-ordering in [16].

```
Definition KFinite (eqA : grel A) (E : gset A) : Prop :=
 ∃ X, (∀ x:A, E x ↔ INA eqA X x).
Definition gpred_list (E: gset A ) : gset (seq A) :=
 fix aux l : Prop :=
  match l with
  | nil ⇒ True
  | x :: xs ⇒ E x ∧ aux xs
end.
```

Fig. 2. Definition of Kuratowski finite and gpred_list

```
Definition E_compat (eqA : grel A) (E:gset A) := ∀ x y, eqA x y → E x → E y.
Lemma Bar_gpred_list : ∀ (eqA : grel A) (E : gset A),
  E_compat eqA E → IFinite eqA E → IFinite eql (gpred_list E).
```

Fig. 3. Finiteness of gpred_list

Proposition 2. *(Bar_fun) If f is a function preserving equality*

$$\forall x\ y, x =_A y \rightarrow fx =_B fy$$

and if E is inductively finite, then fE, the image of E by f, is also inductively finite.

Both Prop. 1 and 2 are important for the proof of the main Lemma 2.

```
Variable f : A → B.
Variable eqA : grel A.
Variable eqB : grel B.

Definition f_set (E: gset A) : gset B := fun (y:B) ⇒ exists2 x, E x & eqB (f x) y.
Variable f_compat : ∀ (a a': A), eqA a a' → eqB (f a) (f a').
Lemma Bar_fun : ∀ E , IFinite eqA E → IFinite eqB (f_set E).
```

Fig. 4. Definition of the property Bar_fun

A major result for proving Prop. 1 is the fact that if E is inductively finite on A then the relation of strict inclusion (sup) between subsets of E enumerated by a list is well-founded.

Theorem 1. *(IFinite_supwf) For all compatible set E, the relation* sup *is well founded if and only if E is inductively finite.*

Corollary 1. *If E is inductively finite, any monotone operator acting on subsets of E enumerated by a list has a least fixed-point.*

This is proved by building this list by well-founded recursion.

Theorem IFinite_supwf : $\forall$ (eqA : grel A) (E : gset A) , E_compat eqA E $\rightarrow$
(IFinite eqA E $\leftrightarrow$ well_founded sup (gpred_list E)).

Fig. 5. Strict inclusion of lists

2 Regular Expressions

Now that we know how to encode inductively finite sets in type theory, we focus on the main purpose of this paper, deciding regular expressions equivalence. It is direct to represent the type of all regular expressions on a given alphabet Σ as an inductive type. Following Brzozowski's approach, we work with extended regular expressions, having conjunction and negation as constructors, and a "." constructor that matches any letter of the alphabet.

$$E, E_1, E_2 ::= \emptyset \mid \epsilon \mid a \mid . \mid E_1 + E_2 \mid E^* \mid E_1 E_2 \mid E_1 \& E_2 \mid \neg E$$

It is a remarkable feature of Brzozowski's algorithm that it extends directly the treatment of negation. If one uses finite automata instead, the treatment of negation is typically more difficult, since one would have to transform an automaton to a deterministic one in order to compute its complement.

To each regular expression E we associate a boolean predicate (using Ssreflect's pred) $L(E)$ on the set of words Σ^* such that a word u satisfies $L(E)$ if and only if u is recognized by E. So the boolean function $L(E)$ reflects the predicate of being recognized by the language E. We can then write "u \in E" (this is a notation for mem E u) to express that the word u is recognized by E. We consider that two languages are equal if they contain the same words:

$$\forall L_1\ L_2,\ L_1 = L_2 \quad \leftrightarrow \quad \forall u \in \Sigma^*, u \in L_1 = u \in L_2$$

Two regular expressions are equivalent if their associated languages are equal.

It is direct to define a boolean $\delta(E)$ (or has_eps E in our formalization) which tests whether the empty word ϵ is in $L(E)$ or not (see Fig. 6).

```
Variable symbol : eqType.

Definition word := seq symbol.
Definition language := pred word.

Inductive regular_expression :=
  | Void
  | Eps
  | Dot
  | Atom of symbol
  | Star of regular_expression
  | Plus of regular_expression & regular_expression
  | And of regular_expression & regular_expression
  | Conc of regular_expression & regular_expression
  | Not of regular_expression.

Definition EQUIV (E F:regexp) := ∀ s:word, (s \in E ) = (s \in F).
Notation "E ≡ F" := (EQUIV E F) ( at level 30).

Fixpoint has_eps (e: regular_expression) :=
  match e with
  | Void ⇒ false
  | Eps ⇒ true
  | Dot ⇒ false
  | Atom x ⇒ false
  | Star e1 ⇒ true
  | Plus e1 e2 ⇒ has_eps e1 || has_eps e2
  | And e1 e2 ⇒ has_eps e1 && has_eps e2
  | Conc e1 e2 ⇒ has_eps e1 && has_eps e2
  | Not e1 ⇒ negb (has_eps e1)
  end.
```

Fig. 6. Definition of regular expressions and the δ operator

2.1 Derivatives

Given a letter a in Σ and a regular expression E, we define E/a (or der a E), the derivative of E by a, by induction on E (see Fig. 7 for a direct encoding in type theory, or [5] for the original definition). A word u is in $L(E/a)$ if and only if the word au is in $L(E)$: $L(E/a)$ is called the *left-residual* of $L(E)$ by a. It is then possible to define E/u (or wder u E) for any word u by recursion on u

$$E/\epsilon = E \qquad E/(au) = (E/a)/u$$

The function δ and derivation operators give us a way to check whether a word is recognized by a regular expression. With the previous definitions, a word u is in $L(E)$ if and only if ϵ is in $L(E/u)$, which is equivalent to $\delta(E/u) = true$.

```
Fixpoint der (x: symbol) (e: regular_expression) :=
  match e with
  | Void ⇒ Void
  | Eps ⇒ Void
  | Dot ⇒ Eps
  | Atom y ⇒ if x == y then Eps else Void
  | Star e1 ⇒ Conc (der x e1) (Star e1)
  | Plus e1 e2 ⇒ Plus (der x e1) (der x e2)
  | And e1 e2 ⇒ And (der x e1) (der x e2)
  | Conc e1 e2 ⇒ if has_eps e1 then Plus (Conc (der x e1) e2) (der x e2)
    else (Conc (der x e1) e2)
  | Not e1 ⇒ Not (der x e1)
  end.

Fixpoint wder (u: word) (e: regular_expression) :=
  if u is x :: v then wder v (der x e) else e.

Fixpoint mem_der (e: regular_expression) (u: word) :=
  if u is x :: v then mem_der (der x e) v else has_eps e.

Lemma mem_derE : ∀ (u: word) (e: regular_expression), mem_der E u = (u \in E).

Lemma mem_wder : ∀ (u: word) (e: regular_expression),
  mem_der E u = has_eps (wder u E).
```

Fig. 7. Definition of the der operator and some of its properties

2.2 Similarity

Brzozowski proved in [5] that there is only a finite number of derivatives for a regular expression, up to the following rewriting rules:

$$E + E \sim E \qquad E + F \sim F + E \qquad E + (F + G) \sim (E + F) + G$$

This defines a decidable equality over regular expressions, called similarity, which also satisfies $L(E) = L(F)$ if $E \sim F$.

The exact implementation of these rewriting rules is not relevant to show that the set of derivatives is inductively finite. We provide two implementations in our formalization, one exactly matching these three rules, and a more efficient one which also includes the following rules:

$$\begin{array}{lll} E + \emptyset \sim E & (EF)G \sim E(FG) & E \,\&\, E \sim E \\ E \,\&\, \emptyset \sim \emptyset & E \,\&\, (F \,\&\, G) \sim (E \,\&\, F) \,\&\, G & E^{**} \sim E^{*} \\ E \,\&\, \top \sim E & E \,\&\, F \sim F \,\&\, E & \neg\neg E \sim E \\ \epsilon E \sim E\epsilon \sim E & & \emptyset^{*} \sim \epsilon^{*} \sim \epsilon \end{array}$$

The regular expression $\top$ stands for the regular expression that recognize any word, which we implemented as $\neg\emptyset$.

Our implementation is close to the one in [15]. To enforce these additional simplifications, we introduce a notion of canonical form (with a boolean predicate wf_re for being "a well-formed canonical expression") and a normalization function canonize in such a way that $E \sim F$ is defined as canonize E = canonize F (where = is the structural equality). This function relies on the use of smart-constructors which perform the previous rewriting rules. For example, the rewriting rules of Plus are enforced by keeping a strictly sorted lists of all the regular expressions linked by a "+". We then prove that these smart-constructors indeed satisfy the similarity requirements (see Fig. 8). In [11], the idea of normalizing regular expression to enforce the rules is also used, with the exact same idea of keeping sorted lists of regular expression. However, they do not create a different structure and just modify the existing regular expressions.

```
Fixpoint canonize c : canonical_regexp := match c with
  | Void ⇒ CVoid
  | Eps ⇒ CEps
  | Dot ⇒ CDot
  | Atom n ⇒ CAtom n
  | Star c' ⇒ mkStar (canonize c')
  | Plus c1 c2 ⇒ mkPlus (canonize c1) (canonize c2)
  | And c1 c2 ⇒ mkAnd (canonize c1) (canonize c2)
  | Conc c1 c2 ⇒ mkConc (canonize c1) (canonize c2)
  | Not c1 ⇒ mkNot (canonize c1)
end.

Lemma mkPlusC : ∀ r1 r2, mkPlus r1 r2 = mkPlus r2 r1.

Lemma mkPlus_id : ∀ r, wf_re r → mkPlus r r = r.

Lemma mkPlusA : ∀ r1 r2 r3, wf_re r1 → wf_re r2 → wf_re r3 →
  mkPlus (mkPlus r1 r2) r3 = mkPlus r1 (mkPlus r2 r3).
```

Fig. 8. canonize function and some properties of the Plus smart constructor

Brzozowski's result [5] that any regular expression has only a finite number of derivatives is purely existential. It is not so obvious how to extract from it a computation of a list of all derivatives up to similarity: even if we have an upper bound on the number of derivatives of a given regular expression E, it is not clear when to stop if we start to list all possible derivatives of E. In type theory, this will correspond to the fact that we can prove that the set of all derivatives of E is *inductively* finite up to similarity, but we can not prove without a further hypothesis on similarity (namely that similarity is closed under derivatives) that this set of *Kuratowski* finite up to similarity. On the other hand, we can always show that the set of derivatives is Kuratowski finite up to equivalence.

3 Brzozowski Main Result

The key property of Brzozowski we need to build the decision procedure is the fact that the set of derivatives of a regular expression is inductively finite (with respect to similarity). An interesting point is that we do not actually need Σ to be finite to prove this fact. However, we need Σ to be finite in order to effectively compute all the derivatives and compare two regular expressions.

The proof uses the following equalities about derivatives (see [5], Annexe II for a detailed proof):

$$(E + F)/u = E/u + F/u \qquad (E \,\&\, F)/u = E/u \,\&\, F/u \qquad \neg(E/u) = (\neg E)/u$$

If $u = a_1 \dots a_n$

$$(EF)/u = (E/u)F + \delta(E/a_1 \dots a_{n-1})F/a_n +$$

$$\delta(E/a_1 \dots a_{n-2})F/a_{n-1}a_n + \dots + \delta(E)F/a_1 \dots a_n$$

and finally

$$E^*/u \sim (E/u)E^* + \Sigma\ \delta(E/u_1) \dots \delta(E/u_{p-1})(E/u_p)E^*$$

for any decomposition of u in non-empty words $u = u_1 \dots u_p$.

We represent the set of all derivatives of a given regular expression E by the predicate

$$\mathcal{D}er\ E = \{F \mid \exists u : \Sigma^*, E/u \sim F\}$$

We proved formally that this set was *inductively* finite with respect to similarity.

Lemma 2. The set of derivatives is inductively finite

For any regular expression E, the set $\mathcal{D}er\ E$ is inductively finite with respect to the similarity:

$$\forall(E : \mathsf{regexp}), \mathsf{IFinite}\ \sim\ (\mathcal{D}er\ E)$$

Proof. The proof is done by induction on E, with a combination of the lemmas Bar_gpred_list and Bar_fun described in Sect. 1. We only describe here the case for Conc, which is the most difficult case. All the other ones are done in a similar way.

By induction, we know that $\mathcal{D}er\ E$ and $\mathcal{D}er\ F$ are inductively finite, and we want to prove that $\mathcal{D}er\ EF$ is too. Equality of regular expressions is performed using the $\sim$ operator with its extension $[\sim]$ to list of regular expressions [regexp]. Let us consider the following function:

$$fConc : \mathsf{regexp} \times \mathsf{regexp} \times [\mathsf{regexp}] \to \mathsf{regexp}$$
$$fConc\ (e, f, L) = e\ f + L_1 + \dots + L_n$$

– Using the equalities of derivatives we just stated, we first show that

$$\mathcal{D}er(EF) \subseteq fConc\ (\mathcal{D}er\ E, \{F\}, [\mathcal{D}er\ F]) \quad (1)$$

- The set $[\mathcal{D}erF]$ is inductively finite for $[\sim]$ thanks to lemma Bar_gpred_list, and the singleton set $\{F\}$ is obviously inductively finite for $\sim$.
- Using Brzozowski minimal set of rewriting rules, it is direct to show that $fConc$ preserves equality:

$$\forall e\, e'\, f\, f'\, l\, l', e \sim e' \wedge f \sim f' \wedge l\,[\sim]\, l' \rightarrow fConc\ (e, f, l) \sim fConc\ (e', f', l')$$

Then the image of the set $\mathcal{D}er\ E \times \{F\} \times [\mathcal{D}er\ F]$ by $fConc$ is inductively finite thank to lemma Bar_fun. Thanks to Lemma 1 and (1), we can conclude that $\mathcal{D}er(EF)$ is inductively finite.

To simplify we assume that Σ is now the type with two elements $\{0, 1\}$, but it would work with any finite set. The particular instance of **regular_expression** over this alphabet is named regexp.

As we said, it is not possible in the general case to enumerate any inductively finite set with a list, but in this particular case, it is possible to do so.

Lemma 3. Enumeration of the set of derivatives
For any regular expression E, it is possible to build a list L_E such that:

- $L_E \subseteq \mathcal{D}er\ E$
- $E \in L_E$
- $\forall(e : \mathit{regexp})(e \in L_E)\, \exists(e' : \mathit{regexp}), (e' \in L_E) \wedge (e' \sim e/0)$
- $\forall(e : \mathit{regexp})(e \in L_E)\, \exists(e' : \mathit{regexp}), (e' \in L_E) \wedge (e' \sim e/1)$

To build such a list, and prove Lemma 3, we apply Coro. 1 on the mononotone function:

$$\mathsf{deriv}\ l = \mathsf{map}(\mathsf{der}\ 0)\ l +\!\!+ \mathsf{map}(\mathsf{der}\ 1)\ l$$

The list L_E is the least fixpoint of deriv above the singleton list $[E]$.

As a consequence of these properties, we can show that any derivative of E is represented inside L_E:

Theorem 2. The list L_E is complete
For any regular expression E and any word u, there is a regular expression e in L_E such that $L(E/u) = L(e)$.
Another way to state it is to say is that the set of all all derivatives of E is KFinite up to equivalence.

Proof. The proof goes by induction on the length of the word u:

- If the length of u is 0, then $u = \epsilon$, and we have $E/\epsilon = E$. We can close this case since E is in L_E.
- If the length of u is $n + 1$, then $u = vi$ where its last letter i is either 0 or 1. By induction, there is e in L_E such that $L(e) = (E/v)$. Using the two last properties of L_E as described in the previous lemma, there is e' in L_E such that $e' \sim e/i$, which implies $L(e') = L(e/i)$. If we combine both conclusions, we get that $L(E/u) = L((E/v)/i) = L(e/i) = L(e')$, which ends this proof.

What we prove is that the set of all derivatives is Kuratowski finite up to equivalence. Contrary to what one may thought at first, it does not seem possible to

show that this set is Kuratowski finite up to similarity. In order to be able to prove it, we need a priori a stronger condition on $\sim$, that we have $A/0 \sim B/0$ and $A/1 \sim B/1$ whenever $A \sim B$. This is the case for Brzozowski minimal set of rules, but it is not the case for our efficient implementation of the similarity. (As it turned out, to have a list up to equivalence is sufficient to get a decision for equivalence.) In particular, the rule $E^{**} \sim E^*$ is not stable by derivation, it would require to add $E^*E^{**} \sim E^*$ to our set of rules.

4 Description of the Decision Procedure

From the definition of regular expressions equivalence $\equiv$ and the basic properties of the δ operator, we can derive another specification for being equivalent:

$$\begin{aligned} \forall E\ F,\ E \equiv F\ &= L(E) = L(F) \\ &\leftrightarrow \forall u \in \Sigma^*, u \in L(E) = u \in L(F) \\ &\leftrightarrow \forall u \in \Sigma^*, \delta(E/u) = \delta(F/u) \end{aligned}$$

```
Definition delta2 (ef:regexp × regexp) := let (e,f) := ef in has_eps e == has_eps f.

Definition build_list_fun : regexp → regexp → seq (regexp× regexp).

Definition regexp_eq (r1 r2: regexp) : bool := (all delta2 (build_list_fun r1 r2)).

Lemma regexp_eqP : ∀ (r1 r2:regexp), reflect (r1 ≡ r2) (regexp_eq r1 r2).
```

Fig. 9. Decision procedure with its correctness proof

For any regular expressions E and F, we consider the set

$$\mathcal{D}er_2\ E\ F = \{(e, f) \mid \exists (u : word), e \sim E/u \wedge f \sim F/u\}$$

This set is included inside $\mathcal{D}er\ E \times \mathcal{D}er\ F$, so with Lemmas 1 and 2, we can conclude that $\mathcal{D}er_2\ E\ F$ is inductively finite for any $E\ F$. A similar approach to the proof Lemma 3 and Thm. 2 allows us to conclude that $\mathcal{D}er_2\ E\ F$ can be enumerated by a list $L_{E,F}$ and for all word u in Σ^*, there is (e, f) in $L_{E,F}$ such that $L(e) = L(E/u)$ and $L(f) = L(F/u)$.

This property of $L_{E,F}$ is enough to decide the equivalence: we know that $L(E) = L(F) \quad \leftrightarrow \quad \forall u \in \Sigma^*, \delta(E/u) = \delta(F/u)$ and we proved that, for any u in Σ^*, there is (e, f) in L_{EF} such that $L(e) = L(E/u)$ and $L(f) = L(F/u)$. Since $\delta(e) = \delta(E/u)$ and $\delta(f) = \delta(F/u)$, we can show that

$$L(E) = L(F) \quad \leftrightarrow \quad \forall (e, f) \in L_{EF}, \delta(e) = \delta(f)$$

which is a decidable predicate.

5 Representation in Type Theory

While we have been carrying out our formal development in the systems Coq [7] and Ssreflect [18], we never use in an essential way the impredicativity of the sort of propositions. So all our proofs could have been done as well in a predicative system with universes, such as the one presented in [12], extended with inductive definitions.

One key issue is for the representation of the function regexp_eq which is defined by well-founded recursion, itself defined (see Fig. 10) by saying that all elements are accessible [14]. Indeed, this function is defined by recursion on the fact that the set of derivatives of a regular expression is inductively finite, which can be expressed, as we have seen above, by the fact that a relation is well-founded. This representation, theoretically sound, is problematic operationally: the computation of this function requires a priori the computation of the proof that an element is accessible. This computation is heavy, and furthermore seems irrelevant, since the accessibility predicate has only one constructor. In order to solve this problem, we follow the solution of Barras, refined by Gonthier, which allows us to keep the logical soundness of the representation with a satisfactory computational behaviour. The main idea is to "guard" the accessibility proofs by adding constructors (see Fig. 10). If we replace a proof that a relation is well-founded wf by its guarded version guard 100 wf, we add in a lazy way 2^{100} constructors, and get in this way a new proof that the relation is well-founded which has a reasonable computational behavior.

```
Inductive Acc (E:gset A) (R:grel A) (x:A) : Prop :=
  Acc_intro : (∀ y, E y → R y x → Acc E R y) → Acc E R x.

Definition well_founded (E : gset A) (R : grel A) := ∀ a: A, Acc E R a.

Fixpoint guard (E: gset A) (R : grel A) n (wfR: well_founded E R):
  well_founded E R :=
  match n with
    | 0 ⇒ wfR
    | S n ⇒ fun x ⇒ Acc_intro (fun y _ _ ⇒ guard E R n (guard E R n wfR) y)
  end.
```

Fig. 10. Guarded version of accessibility proof

6 Some Examples

One important feature of our formalization is the fact that we obtain the decision procedure as a type theoretic boolean function, with which you can compute directly without extracting to ML code. We can then use this function to build in type theory other tactics to solve problems which can be encoded in the language of regular expressions.

The following tests have been performed on an Intel Core2 Duo 1.6 GHz, with 2 GB of memory, running `Archlinux` with kernel 2.6.39. We used Coq v8.3pl2 and Ssreflect v1.3pl1 to formalize the whole development. It is direct to reduce the problem of inclusion to the problem of equivalence by expressing $E \subseteq F$ as $E + F \equiv F$.

The first example we tested is due to Nipkow and Krauss in [11]. The theories of Thiemann and Sternagel [20] contains the lemma which reduces to the following inclusion of regular expressions

$$0(00^*1^* + 1^*) \subseteq 00^*1^*$$

Our implementation of similarity answers `true` in 0.007 seconds.

The second example is extracted from the following predicate

$$\forall n \geqslant 8, \exists x\ y, n = 3x + 5y$$

Is this predicate true or false ? This can be rewritten as the following regular expression problem:

$$000000000^* \subseteq (000 + 00000)^*$$

Our implementation answers `true` in 1.6 seconds.

Some more examples can be found in the file `ex.v` at [8]. Since we are only looking for building a tactic on top of this decision procedure, like in the reference [11], both results are within acceptable range for this purpose.

Conclusion and Future Work

The main contributions of this work are

- a complete formalization in type theory of Brzowoski's algorithm for testing the equivalence of regular expressions
- a new definition of finiteness and formal proofs of its basic closure properties, which may have an interest in itself
- the experimental verification that it is feasible to define in type theory functions by well-founded induction and to prove their properties, obtaining programs that have a reasonable operational behavior[3]

As a direct extension of Brzozowski's procedure, we also defined and proved correct a decision algorithm for the inclusion of regular expressions, that we have tested on some simple examples.

While doing this formalization, we discovered two facts about Brzozowski's algorithm that may not be obvious at first, and which are examples of what one

[3] As far as we know, this approach to representation of terminating general recursive function in type theory has not been tested before. For instance this approach is explicitly rejected in the reference [4], as being "inconvenient", since it "requires mixing some non-trivial proofs with the codes" while our work shows that it is reasonable in practice and theoretically satisfactory.

may learn from formalization (and new to us, though we have been teaching the notion of Brzozowski's derivatives for a few years). First, the number of derivatives is finite even if the alphabet is not. (However in practice, one has to restrict to finite alphabets if one wants to extract the list describing the derivatives.) Second, it is not so obvious how to extract from Brzozowski's purely existential result an actual computation of a list of all derivatives up to *similarity* (as one may have expected at first; without the further assumption that similarity is closed under derivatives we obtain only a list of derivatives up to equivalence).

There are other notions of derivatives that are worth investigating, like in [21] where they use partial derivatives known as *Antimorov*'s derivatives. A natural extension of this work would be, like in the references [4,11] to use it for a reflexive tactic for proving equalities in relation algebra. We don't expect any problem there, following [4,11]. A more ambitious project will be to use this work for writing a decision procedure for the theory WS1S [6], where formulae of this language are interpreted by regular expressions. Since we use extended regular expression, we have a direct interpretation of all boolean logical connectives, and what is missing is the interpretation of the existential quantification. For giving this interpretation, one possible Lemma would be to show that any extended regular expression is equivalent to a regular expression which uses only the operator of union, concatenation and Kleene star. This in turn should be a simple consequence of the fact that the set of derivatives of a given expression is Kuratowski finite up to equivalence. Using this result, we can then define given any map $f : \Sigma_1 \to \Sigma_2$ extended to words $f^* : \Sigma_1^* \to \Sigma_2^*$, and given a regular expression E over Σ_1, a new regular expression $f^*(E)$ over Σ_2 such that $L(f^*(E)) = f^*(L(E))$. It is then possible to interpret existential quantification using this operation.

References

1. Almeida, J.B., Moreira, N., Pereira, D., de Sousa, S.M.: Partial Derivative Automata Formalized in Coq. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 59–68. Springer, Heidelberg (2011)
2. Bezem, M., Nakata, K., Uustalu, T.: On streams that are finitely red (submitted, 2011)
3. Braibant, T., Pous, D.: A tactic for deciding Kleene algebras. In: First Coq Workshop (August 2009)
4. Braibant, T., Pous, D.: An Efficient Coq Tactic for Deciding Kleene Algebras. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 163–178. Springer, Heidelberg (2010)
5. Brzozowski, J.A.: Derivatives of regular expressions. JACM 11(4), 481–494 (1964)
6. Büchi, J.R.: Weak second order arithmetica and finite automata. Zeitscrift Fur Mathematische Logic und Grundlagen Der Mathematik 6, 66–92 (1960)
7. The Coq Development Team, `http://coq.inria.fr`
8. Coquand, T., Gonthier, G., Siles, V.: Source code of the formalization, `http://www.cse.chalmers.se/~siles/coq/regexp.tar.bzip2`
9. Coquand, T., Spiwack, A.: Constructively finite? In: Laureano Lambán, L., Romero, A., Rubio, J. (eds.) Scientific Contributions in Honor of Mirian Andrés Gómez Servicio de Publicaciones, Universidad de La rioja, Spain (2010)

10. Johnstone, P.: Topos theory. Academic Press (1977)
11. Krauss, A., Nipkow, T.: Proof Pearl: Regular Expression Equivalence and Relation Algebra. Journal of Automated Reasoning (March 2011) (published online)
12. Martin-Löf, P.: An intuitionistic type theory: predicative part. In: Logic Colloquium 1973, pp. 73–118. North-Holland, Amsterdam (1973)
13. Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Engineering Cybernetics 5, 51–57 (1996)
14. Nordström, B.: Terminating general recursion BIT, vol. 28, pp. 605–619 (1988)
15. Owens, S., Reppy, J., Turon, A.: Regular-expression Derivatives Re-examined. Journal of Functional Programming 19(2), 173–190
16. Richman, F., Stolzenberg, G.: Well-Quasi-Ordered sets. Advances in Mathematics 97, 145–153 (1993)
17. Russell, B., Whitehead, A.N.: Principia Mathematica. Cambridge University Press (1910)
18. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. Journal of Formalized Reasoning 3(2), 95–152 (2010)
19. Tarski, A.: Sur les ensembles finis. Fundamenta Mathematicae 6, 45–95 (1924)
20. Thiemann, R., Sternagel, C.: Certification of Termination Proofs Using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)
21. Wu, C., Zhang, X., Urban, C.: A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl). In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 341–356. Springer, Heidelberg (2011)